Division of Informatics, University of Edinburgh

A Web Based Replayer For Proof General

by

Jonathan Freear

Informatics Research Report EDI-INF-RR-0027

Division of Informatics **September 2000**
http://www.informatics.ed.ac.uk/

A Web Based Replayer

For Proof General

MSc Project Report

Jonathan Freear

September 19, 2000

Abstract

Proof General is a generic interface for proof assistants, based on Emacs. It has been developed at the LFCS in the University of Edinburgh. One of the nice features of Proof General is that it is very easy to replay existing proofs, by mouse clicks alone. No low level understanding of a proof assistant is needed to step through proofs. The aim of this project is to have a web-based version of Proof General which allows for this proof replay, running a proof assistant remotely. The main aspect is to implement an engine for script management (colouring of lines of files), displaying in a web browser, sending lines to a proof assistant process and displaying the results.

Acknowledgements

I would like to thank my supervisor, David Aspinall, for his extreme patience and expert guidance throughout the course of this project.

Contents

Chapter 1

Introduction

This project is concerned with the development of a web-based replayer for Proof General, allowing for the step through of a specified, but extendable set of proofs remotely through a web browser. The aim of this report is to present an accurate and detailed account of the project.

In the following introduction I will describe the aims and objectives of the project, the motivating factors behind them and some of the constraints involved in developing the project before providing a summary of the content of the subsequent chapters.

1.1 Project Aims & Objectives

The goal of the project is to implement a web-based application allowing for the replaying of various proof scripts for a number of the different proof assistants supported by Proof General. This application will aim to provide the same script management features as Proof General itself, described in greater detail in chapter two.

1.2 Motivating Factors

The motivation behind this project is to provide the user with a means of replaying proofs remotely without the need for installation of either Proof General or any of the supported proof assistants.

This will allow many of the users who currently use the basic command line interface of the proof assistants to learn more about the Proof General Project without the cost of downloading and installing all of the required elements.

1.3 Development Environment

This project was developed on my home machine - a standard PC running Red Hat 6.2. The software was developed with the Java 2 Development Kit from Sun Microsystems, with two additional packages, gnu.regexp, from the Gnu Software Foundation (required as Java 2 provides no support for regular expressions) and Suns Servlet Development Kit to develop the server side code. The servlets were initially run on my home machine in a servlet container provided by the Jakarta-Tomcat package of the Jakarta project.

All of the supported proof assistants were also installed on my home system.

1.4 Document Structure

This section presents a summary of the remaining chapters in the report.

- Background

In the next chapter, I will outline the background work which enabled me to approach this project, and highlight some of the more important elements of Proof General and the Proof General Project.

- Design

In Chapter Three I analyse the requirements of the Replayer and detail and explain the design decisions which were made to fulfill these objectives.

- Implementation

This Chapter serves to highlight some of the implementation details and provides greater detail of some of the more important aspects of the code.

- Testing & Evaluation

Chapter Five represents a brief look at the testing which was carried out, and demonstrates some of further elements to be evaluated before the system could be considered finished.

- Conclusion

Finally I discuss the project results, scope for future work and my own view of the project as a whole.

Appendices are also included containing code for the Replayer application and a sample of Javadoc generated documentation for some of the more important classes.

Chapter 2

Background

This Chapter aims to provide a more detailed introduction into the Proof General project, introducing and explaining the crucial ideas and terminology critical to the comprehension of report and the project as a whole.

2.1 Proof General

Proof General is a generic interface for interactive proof assistants, based on Emacs.

Proof assistants are systems which have been developed to perform various tasks required in the development of formalized proofs. Formal proofs are particularly important in computer science as the correctness of both software and hardware is nothing more than a mathematical statement capable of confirmation by formalization.

Proof assistants can not only help in checking sizable formalized proofs but can also help with coming up with formalized proofs in the first place by carrying out various tasks such as providing tactics and decision procedures, keeping track of assumptions and variables and providing libraries of definitions and previously proved theorems. As would be expected, different proof assistants have different strengths and weaknesses.

Proof General currently supports the following proof assistants: Lego, Coq and Isabelle, three systems based on type theory which happen to have weak user interfaces. Support for HOL98 is also currently underway. Proof General itself is a highly extendable project, capable of adaptation to include support for other proof assistants with great ease.

The main idea behind Proof General, and its greatest strength, is its genericity. As Aspinall explains[1]:

> [Proof General] exploits the deep similarities between systems by hiding some of their superficial differences. Just as a web browser presents a similar interface to different protocols - http, ftp or file, so Proof General presents a similar interface to different proof assistants.

Many of the proof assistants still employ a simple command line interface and Proof General aims to provide a middle ground between this basic approach and that taken by the sophisticated GUI alternatives; Proof General remains largely text based, but still provides many powerful features, the most relevant of which are listed below.

Features[1]

Script management connects the editing of a proof script (discussed below) directly to an interactive proof process, maintaining consistency between the edit buffer and the state of the proof assistant. Proof General colors a proof script to show the state in the proof assistant. Parts of a proof script that have been processed are displayed in blue and are "locked" – they cannot be edited. Proof General has commands for processing new parts of the buffer, or undoing previously processed parts. Issuing commands directly from a proof script file saves on tedious cutting and pasting from a file to the shell command line.

[1] Details taken from the Proof General website, *http://www.dcs.ed.ac.uk/~proofgen/*, currently hosted at *http://www.zermelo.dcs.ed.ac.uk/~proofgen/index.phtml?page=features*.

The screen shot below provides an example of this script management in action.

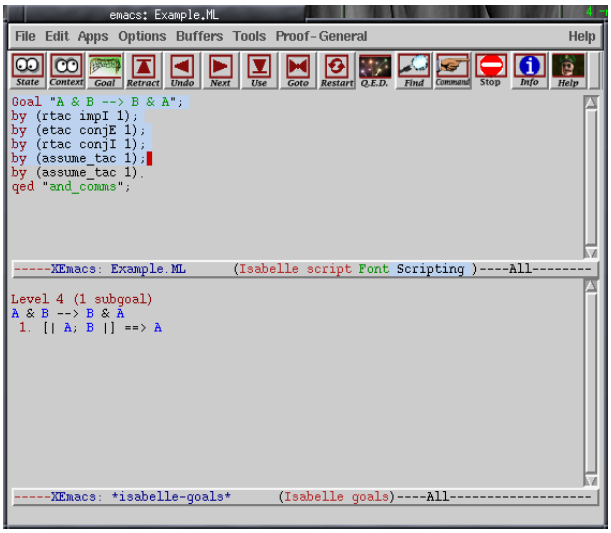


Figure 2.1: Proof General Screen Shot

As mentioned above, Proof General is designed for proof assistants which have a command-line shell interpreter. When using Proof General, the proof assistant's shell is hidden from the user resulting in **simplified communication**. Communication takes place via three buffers (Emacs text widgets). The script buffer (the top section of the screenshot) holds input, the commands to construct a proof. The goals buffer (the lower section) displays the current list of subgoals to be solved. The response buffer displays other output from the proof assistant. By default, only two of these three buffers are displayed at once. This means that the user only sees the output from the most recent interaction, rather than a screen full of output from the proof assistant.

Proof General has a **toolbar** with buttons for examining the proof state, starting a proof, manoeuvring in the proof script, restarting the prover, saving a proof, searching for a theorem, issuing a command, interrupting the assistant, and getting help. Using the toolbar, it is possible to replay proofs without any knowledge of the low-level commands of the proof assistant (or any Emacs hot-keys).

Proof General employs **syntax highlighting** (an editing feature which decorates a file with different colors or fonts depending upon the syntax of the language in use), decorating proof scripts: proof commands are highlighted - for example, different fonts may be used for definitions and assumptions, or as in the case above, keywords may be highlighted.

Most importantly, Proof General is **generic**; it can be adapted to a new proof assistant with surprisingly little effort. Adapting for a new proof assistant is mainly a matter of setting some variables with regular expressions to help parse output from the prover, and setting other variables with commands to send to the prover. It is also extremely simple to customize Proof General to suit personal preferences for all of the supported proof assistants.

2.2 Proof Script

Since the Replayer is designed to replay previously constructed proofs only, the idea of a proof script is central to this project. A *proof script* is a sequence of commands which constructs definitions, declarations, theories and proofs in a proof assistant. For the purposes of this project all proof scripts will have been saved in a text file and will represent full, complete proofs.

Commands can be sent from a proof script to the proof assistant in exactly the same way as if a user was entering commands at a proof assistants shell interface; similarly the system responds at each step (for example, with a new list of subgoals or a failure report).

It should be noted that a proof script can hold more than one proof and that the scripts can also contain comments within special markers which the proof assistant ignores. Figure 2.2 highlights these features along with terminal characters - delimiters between separate commands in a proof - and the Goal command and Save command - commands which instruct the proof assistant of the start of a new proof and the saving of a completed proof respectively.

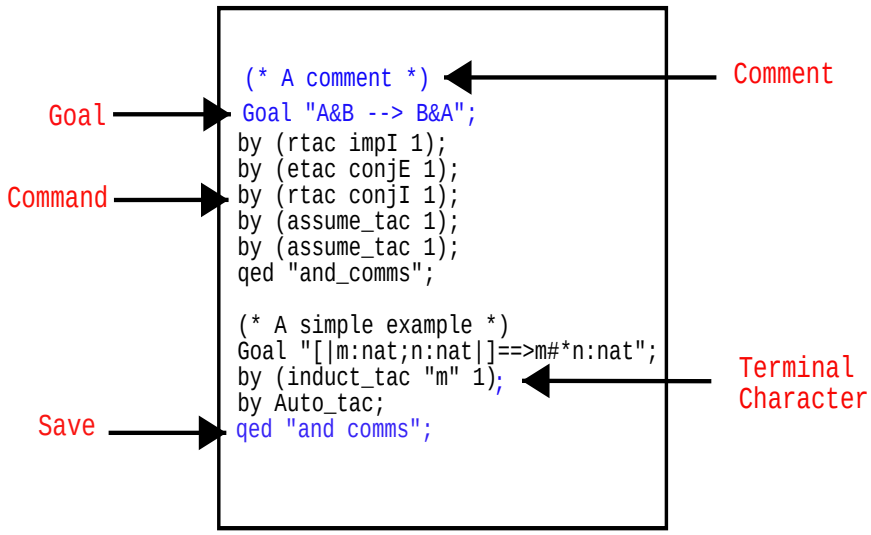


Figure 2.2: An Example Proof Script for Isabelle

2.3 Required Features

The Replayer does not require all of the features of Proof General. However some of the more fundamental features mentioned above are necessary. In particular, since the main purpose is to support the replaying of proof scripts, those features which make script replaying so simple in Proof General should be supported by the replayer.

These features include the Script Management, utilizing the file highlighting; the simple communication between the proof script and the proof assistant, including the output monitoring; the navigation toolbar for intuitive movement through the script; and the genericity of the original Proof General system - the Replayer should present the same user interface for each of its supported proof assistants and should also be extendable to support future proof assistants as they are added to the Proof General project.

Since it will support different proof assistants it should provide a simple means to change proof script file and proof assistant enabling the replaying of different files for different assistants in the same session.

2.4 Getting Started

Most of my background work involved delving into the emacs lisp code of Proof General in order to understand how the system operated and what was required of the Replayer.

Similarly, although I had some experience of a few proof assistants, including Lego, I had to carry out some research into the structure of proof scripts and the syntax employed by the different proof assistants.

On a more personal level, a great deal of time was spent at the beginning and throughout the project on developing my own programming skills. This project represented my first reasonable sized implementation of a system. I had very little experience or knowledge of the Java programming language, and had no experience whatsoever of either servlet technology (or indeed any client/server programming) or even regular expressions which were both required during the course of the project, and as such a great deal of time and effort was expanded acquiring the necessary knowledge.

Chapter 3

Design

The design requires a web based architecture with the client display interacting with a proof assistant running on the server. Therefore, the first design choice I faced was to decide were the majority of the work was going to take place. The options were to have basically the whole system running server side simply sending marked up HTML to a client browser or to rely on an applet running client side to do the majority of the work and to interact with a mediator on the server side to simply pass the commands to the proof assistant and send back the output.

I decided upon the latter design as this seemed to offer greater flexibility and grounds for development of the project - the Replayer could in effect run as a stand alone application, which would allow access to the local file system rather than just the downloading of a particular script; this also escapes the considerable problems and complications of hardwiring the various different options into an HTML page client side.

Naturally the applet was to be implemented in Java, introducing all the usual benefits that Java bestows upon programming including the simplicity of design, the quicker debugging and the easier maintainance of programs written in an object-oriented language; the cross platform portability, the rich class libraries contained in the Java Development Kit and the excellent network capabilities

of Java.[1]

A decision was made to initially design a system to run locally and then to deal with the more technical details of communicating over HTTP once this was up and running and the more conceptual elements had been dealt with. This enabled me to concentrate on the more important parts of the project.

3.1 General System Design

Object-oriented software design is all about objects. An object can be viewed as a black box which receives and sends messages to other objects. The code and data which an object actually contains are not as important as the interface which it presents and through which messages are communicated. In particular, the user of an object need have no idea how this black box goes about its business, as long as it can communicate with the object correctly through its interface.

The first design phase involved looking at the requirements of the system and identifying the main objects involved. Clearly there needed to be a GUI for the user to interact with the system, which I will deal with in the next section. The other objects which immediately stood out were an object to represent the proof script itself, and an object to interact with the proof assistant to which the commands were going to be sent.

- The ProofScript Object

I envisaged the proof script object holding all the crucial information regarding the proof script itself and navigation through it. This meant that it needed to hold a representation of the file being replayed and have a means of knowing the current position within the file. It would also require a simple parser to be able to parse through the script, returning the commands as required. This dynamic processing appeared to be more sensible than the alternative: the construction of

[1] Please see *http://www.java.sun.com/docs/overviews/java/java-overview-1.html* for further discussion.

a more complicated ProofScript object on initialization containing, for example, an array of all the commands and other information required, as firstly, the proof scripts themselves can be very long and secondly the user may not intend to replay the whole proof.

From the black box point of view this meant that it would need to receive messages from the GUI, instructions on where to navigate to, and send messages back to the display, updating the view of the file. It would also need to send messages to the ProofAssistant Interact object - namely the commands which have been parsed.

The ProofScript object is were the script management occurs and is dealt with in detail below in 3.3.

- The ProofAssistant Interact Object

The ProofAssistant Object was to represent the connection to the underlying proof assistant process and as such had to provide the input and output streams required to connect to a subprocess.

Considered as a black box it had to receive commands from the ProofScript; to be able to pass those commands onto the correct proof assistant subprocess and retrieve the corresponding output; and to be able to send messages back - the new output to display.

A greater exploration of the system interactions, including UML diagrams, follows below in 3.4, while figure 3.1 emphasizes the black box point of view.

3.2 GUI Design

It was decided to build the graphical user interface using Swing, the standard windowing toolkit for Java version 1.2. Although Swing is not yet in standard use for web browsers, the intended users of this system are likely to be experienced and advanced computer users who are likely to have more recent versions of the Java plug-ins, or at least access to systems which provide them. Swing

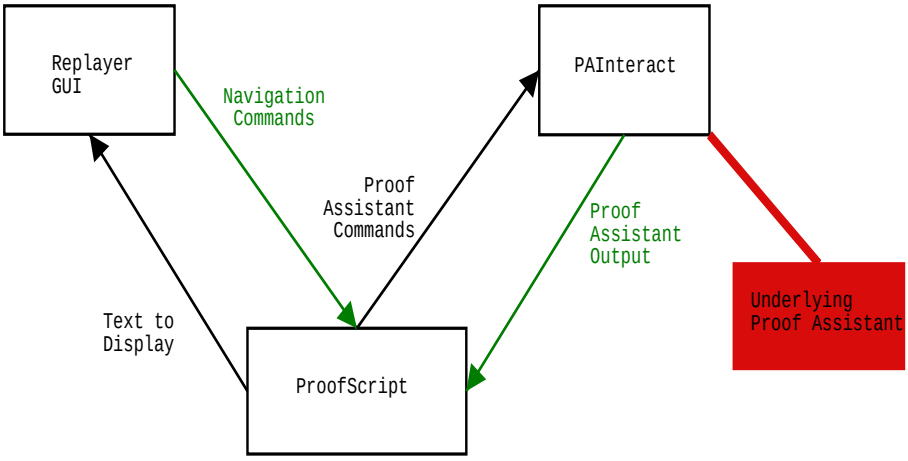


Figure 3.1: Object Interactions from a Black Box Viewpoint

also provides simple means to change an application to an applet, using class javax.swing.JApplet.

Swing allows simple, rapid development of interfaces by providing predefined GUI components and provides various advantages over the previous windowing toolkit for Java, AWT - such as the ability to include more professional features such as tool tips and mnemonics; and most notably its pluggable look-and-feel. The look of a window in a Windows 95/98/NT environment is different from that in a Motif environment running on a Unix workstation. With the Swing API, you can choose the look-and-feel to be that of a particular platform, to be a platform-independent look-and-feel, or to be a look-and-feel that depends on the platform on which the program is running. Swing also provides a rich text framework which greatly simplifies the job of displaying and manipulating text.

The main motivating factor behind my GUI design was simplicity of use. Many proof assistants still have a primitive command line interface and as Aspinall notes[1]:

> expert users often prefer ... and work more effectively with it. This may be for several reasons: because the GUIs are poorly engineered, because they are overly restrictive or do not scale to large develop-

> ments or simply because current experts do not want to change their working practices and waste effort on learning an interface.

It was therefore crucial that the design was as simple, intuitive and user friendly as possible. At the same time I felt that as the Replayer was to be part of the larger Proof General project, it was important to keep the design close to the current interface used by Proof General in Emacs.

The following sections highlight the crucial elements involved in the interface: a file display, a log display, an output display and a means of navigating through the file in the simplest manner possible.

3.2.1 File Display

The area of the interface to display the current proof script - this corresponds to Proof Generals script buffer discussed in section 2.1. As such it needs to be able to display a reasonable amount of text and most importantly be able to highlight the appropriate areas correctly. As the file is extremely likely to be larger than the viewable area of the display, it also needs to scroll with the file display.

Finally, in order to implement the more complicated navigational capabilities the actual display needs to be sensitive to mouse clicks, to be able display a cursor and to be able to connect the position of the cursor in the display with the respective position within the proof file. This enables support for Proof General's GOTO feature, by which it is possible to click on any part of the script buffer and process to that point.

3.2.2 Log Display

A log display was added to the design specification so that the user can keep a track of their session and the proofs replayed; importantly it also gives the user a chance to see the exact commands which are being sent to the proof assistant and all of the output which is received back, as much of this is hidden from the

user.

Although the log output is simple plain text, the display does require the possibility of scrolling as again the output is likely to be greater than the visible area.

3.2.3 Proof Assistant Output

The area of the GUI to display the output from the proof assistant - this is equivalent to a combination of Proof General's goal and response buffer. As the output from the proof assistants is plain text the display is greatly improved and made much more readable by the use of colouring, indentation and the restriction of output to the most salient information. Therefore this text area needs to support all of these operations.

3.2.4 Navigation

In order to navigate through a proof script it is necessary to move backwards and forwards. The choice, however, is by how much. Since proofs are normally walked through step by step it seemed natural to offer options to move forwards and backwards one state at a time. Equally however, this would become tiresome if an extremely long script was being replayed and the point of interest was some way through. Therefore the goto option mentioned above was also included.

The key for the GUI was to provide the simplest and most intuitive means possible for the user to navigate through the script. To this end I decided upon a series of buttons which enable navigation by mouse clicks alone. These buttons provide all the necessary controls to allow movement forwards and backwards. However in an effort to keep the visible interface minimal I decided to place some operations (those used less frequently) in a pop up menu - still accessible by mouse.

A normal menu was also required to keep the interface as typical and intuitive as possible.

Finally, as many users prefer program interaction through the keyboard alone

as opposed to continuously switching between keyboard and mouse, all of the operations should also be accessible by shortcut keys.

In conclusion, since this project is based around a GUI, the design is extremely important. Further aspects of GUI design are discussed in Chapter Five.

3.3 Script Management

The Script Management represents the major conceptual element of the project and involves the following features:

1. Handling the navigation through the script.

2. Colouring the file display to highlight the position in the script.

3. Keeping track of the correct current position.

As already explained above the ProofScript object handles the script management. Below I will explain what is required by the first two items. It should be noted that great care needs to be taken throughout the script management to ensure that the current position within the proof file remains correct and is updated when required.

Throughout the next section the words in bold serve to highlight the methods used.

3.3.1 Script Processing

Movement through the script is possible in two directions - forwards and backwards. In either case it is necessary to first discover the position within the script that you wish to **process to**.

Forwards

Processing forwards is the simpler of the two and I will deal with it first. In order to **process forwards** it is necessary to parse through the script to the required position, keeping hold of the separate commands which have been parsed by **building a queue** of commands. - which need to be accessed in a first-in, first-out order.

These commands then need to be sent to the proof assistant (the **queue is started**) and the **response handled**.

Unfortunately this simple view is complicated somewhat by the needs of processing backwards which I investigate next.

Backwards

To **process backwards** it is necessary to **calculate the command** and then **send that command**.

The difficulty in processing backwards is caused by the necessity of calculating the correct command to send to the proof assistant in order to move back a state.

There are three distinct commands which a proof assistant accepts to undo a proof. Proof General, hides the use of these separate commands and always offers the user an undo option, which then calculates the correct option. In order to understand them it is necessary to understand the concept of being in-proof or out-proof. In-proof simply means that the proof assistant is currently in the middle of a proof - ie a goal has been issued and has yet to be saved.

The three commands are:

1. Undo n.
2. Kill Ref.
3. Forget "id".

Undo n simply moves back n steps in a proof - however not all commands are

undoable. In particular it is only possible to undo commands from within a proof (in-proof) and only as long as that undo does not move the state back past a Goal command (ie out-proof).

Kill Ref enables the proof assistant to move back to before the current goal was issued (moving from in-proof to out-proof).

Forget "id" tells the proof assistant to move back to before the proof entitled "id" and can be used to move backwards from outside a proof. Note that a string id is required to alert the proof assistant to which proof to move back to.

It is sometimes necessary to issue these commands together, in particular a Kill Ref followed by a Forget "id". The effects of these three commands are illustrated in figure 3.2.

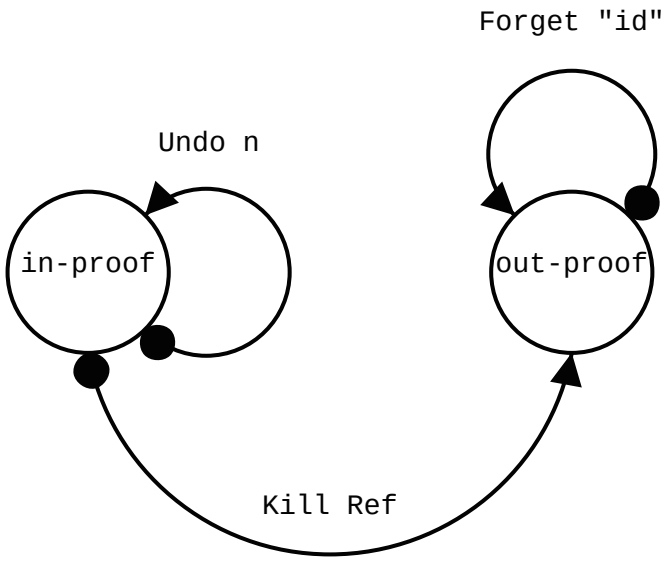


Figure 3.2: Illustration of In / Out Proof

Therefore, to calculate the correct command to send to the proof assistant, dependent on the position within the proof file, it is necessary to hold extra information - a flag to indicate whether the current position is in-proof or out-proof; a record of the string titles of processed goals to enable the use of Forget id; and start positions of processed goals or processed commands (in the current proof) to discover what is at the position we want to process to.

In order to record and keep track of all of this information, following the implementation of Proof General, I felt that a new object was required - a Span.

Spans

Spans are simple objects which are created while processing forwards through a proof file and kept to be searched through for the relevant information when processing backwards - since access is most commonly required to the most recent span it makes sense to store them in a last-in, first-out manner.

They are designed to represent one undoable block of the proof from the proof assistants point of view: either a single command within a proof or a whole proof from goal to save. These undoable blocks are demonstrated in figure 3.3. Basically, each separate command represents a separate block until the whole proof has been constructed and saved, at which point that proof itself is completed and an internal proof state cannot be returned to. Instead the whole proof must be undone ('forgotten') and reconstructed step by step.

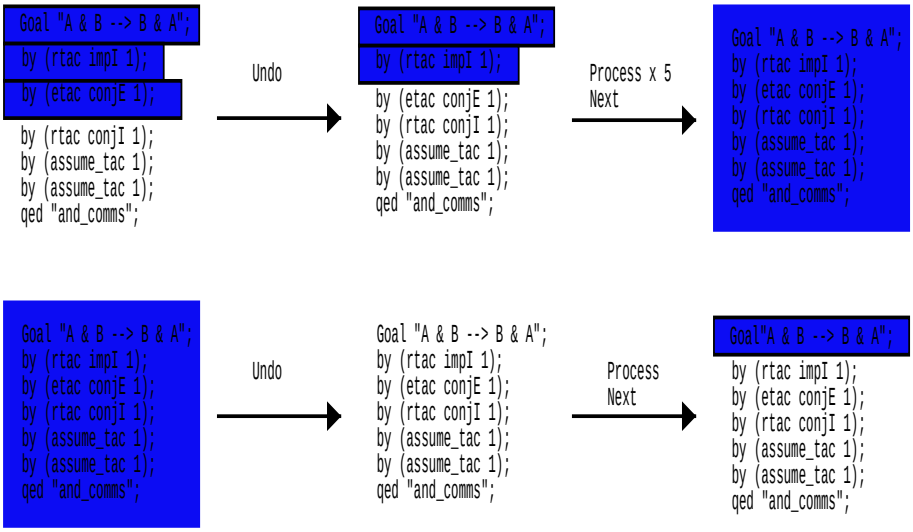


Figure 3.3: Undoable Blocks

In order to work correctly they need to gather the following information when they are constructed: a start and end position corresponding to the relevant

position within the proof file; the 'type' of the span - designed to distinguish between ordinary commands and goal or save commands; and finally a string representing the title (if any) of the current proof. These fields then enable the correct span to be found when searching back through the constructed spans and the correct command to be constructed to move back to the required position.

Finally, it should be noted that the spans are constructed after a response has been received from the proof assistant to ensure that the commands have indeed been processed.

3.3.2 Colouring

A crucial element of the script management is the highlighting of the processed parts of the proof file indicating the current position within the file. This involves calculating which parts of the script have been processed and highlighting them as appropriate.

Fortunately, the design and storage of the Span object makes this task very simple. As the file is represented in the ProofScript object this can be split into two sections at the correct point, one representing the processed section and the other representing what remains. In order to discover this split point, it is suffices to discover the end point of the most recently constructed Span. The **display** in the GUI can then be **updated** to show the change in position.

Figure 3.4 provides a simplified recap of the current system interactions as far as the Script Management is involved. Having looked at what is involved with this, I shall now return to explain more details of the general architecture.

3.4 Generalization

While the above represents the general approach to script management, the whole point of the Proof General project is to provide a common interface for different proof assistants. This section explains how this design goal was fulfilled.

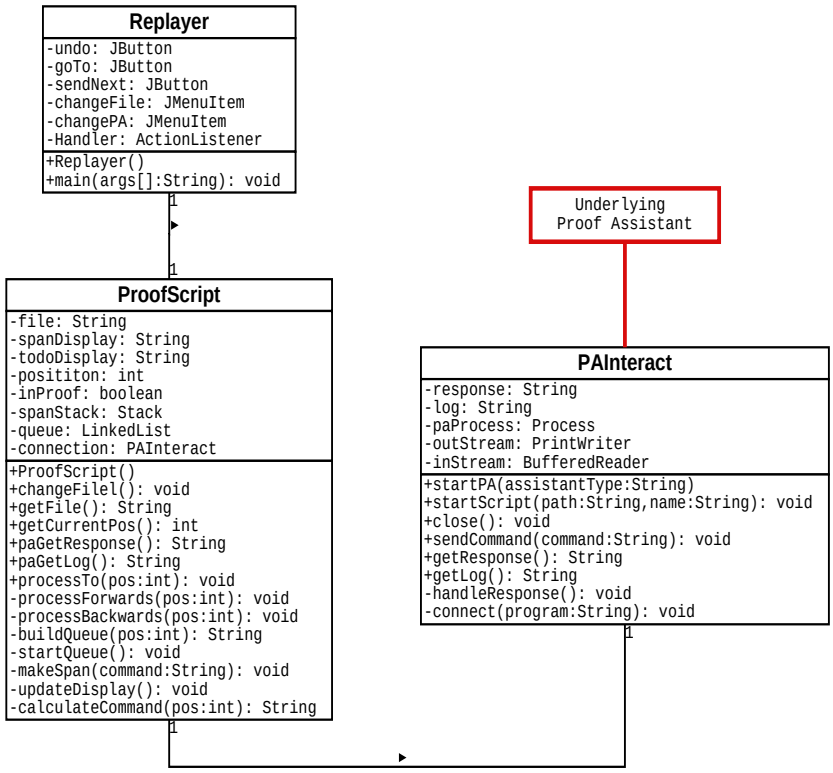


Figure 3.4: A Simplified View of Class Interactions

As mentioned in 2.1 each different replayer requires different commands and has different properties. These commands and properties are required by both the Proof Script object and the ProofAssistant Interact object. For example, Coq has a different 'terminal character' - the character which separates each command of the proof script from one another - to Isabelle and so the parser for the ProofScript object needs to know which terminal character it is searching for. Similarly the exact commands for Kill Ref and Forget "id", the options for the undo command discussed above, are different in Lego and Isabelle; again, ProofAssistant Interact needs to be aware of the correct version to send.

Added to the current system design will be new objects representing the proof assistants (class Isabelle, class Coq, class Lego). At the same time since they are all obviously intimately linked there should be some connection between them. Returning to the black box view of object design commented on earlier it is obvious that the implementation will be greatly eased if each proof assistant implements the same interface (interface ProofAssistant).

Each individual proof assistant object needs to provide information on its particular system, including details such as style of comments, terminal character, goal, save and undo syntax and commands required to initialize the underlying proof assistant subprocess. The system can then utilize the reference to these objects to get the particular commands to send to the prover and by forcing the objects representing the proof assistants to implement a common interface, this ensures that the system will always have the correct information. For example, Isabelle does not actually require the use of ForgetRef to re-open a previously constructed proof; however, the Isabelle object is forced implement the ProofAssistant interface and by returning a powerless command it allows the system to always act in the same fashion.

Just as each proof assistant requires different specific commands, so each one annotates its output with different characters to provide pointers to various information such as the type of output carried or the type of variable. For example, Isabelle provides special characters which notify when a propositional

variable appears in the output and similarly a character which represents the start of the relevant output to be displayed. This output can then be filtered in order to provide the additional features which Proof General offers, such as output highlighting and information selecting; this can be achieved, as it is in Proof General itself, by the use of regular expressions.

Again, the easiest way to do this is with separate objects implementing a common interface (classes IsaFilter, CoqFilter and LegoFilter; interface REFilter). These could easily have been included in the objects implementing ProofAssistant with that interface extended appropriately, but it was felt that by including separate objects in the design the difference between the critical command variables, absolutely crucial to the running of the system and the annotations held in the output information, which serve to provide important but additional, user friendly features such as syntax highlighting, would be stressed.

The class ProofScript and class PAInteract can then get the required information from the particular object which implements interface REFilter (and similarly for ProofAssistant) through that shared interface.

Figure 3.5 presents a simplified class diagram representing interactions between the objects.

3.5 Client/Server

The final design stage involves the splitting of the above system in order to communicate over HTTP with a server, as opposed to the current situation where the underlying proof assistant subprocess resides locally.

The main elements to consider here where:

- to decide whereabouts the system should be split;
- to change the application to an applet;
- to find a means for the client side to contact the server;
- to discover a means for the server to keep track of the different users;

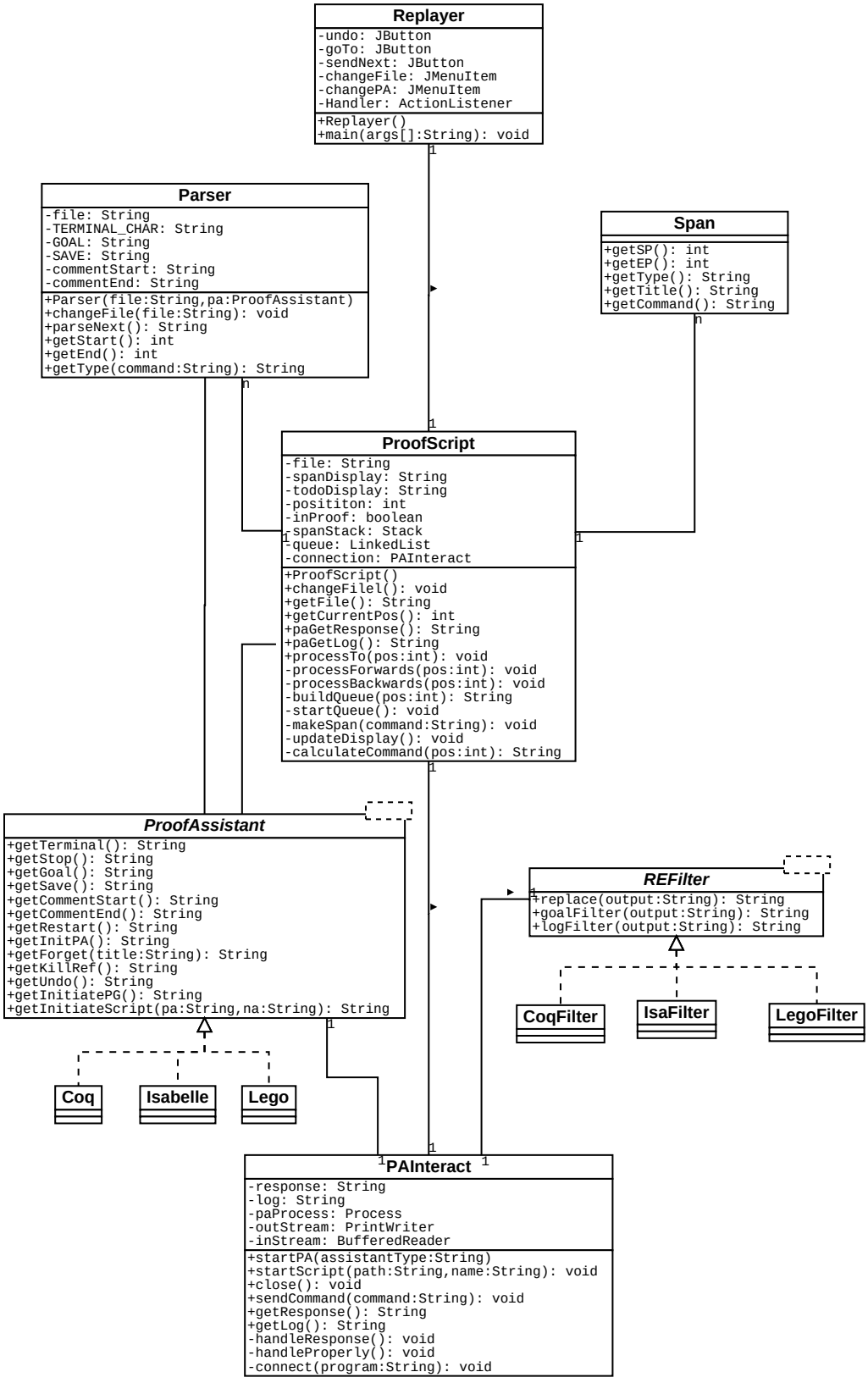


Figure 3.5: Class Diagram

- to connect the different users to their respective proof assistants.

In order to solve these problems some background knowledge was required on Internet communication.

3.5.1 Servlets

Servlets provided the ideal solution to the above problems for a variety of reasons. Importantly, from a practical point of view, they enable the server side code to be written in Java (infact servlets are very much the server side equivalent of applets) - a considerable benefit for this project as time was proving to be a very scarce commodity by the time I reached this stage. Secondly servlet technology has primarily been designed for use with the HTTP protocol, the selected means of communication for this project, and the javax.servlet.HTTP package provides a lot of the required classes. Thirdly, servlets are supported by most major Web servers, including Apache, the World Wide Web Consortium's Jigsaw Web server, Netscape's servers and Microsoft's Internet Information Server.

3.5.2 HTTP

The HTTP protocol uses URLs (Uniform Resource Locators) to locate data on the Internet. Communication between clients and servers via HTTP takes the form of requests and responses. The client sends an HTTP request to the server who responds with an HTTP response. The combination of the Java packages java.net and javax.servlet.HTTP provide all the necessary classes to enable this communication.

3.5.3 Servlet Design

It was decided to keep as much of the system as possible on the client side, as originally envisaged, and, in effect to simply lengthen the communication streams between PAInteract and the subprocess described in 3.1. The benefits

of this include keeping the server load as light as possible and remaining as close as possible to the original design.

Still some rearrangement of classes is required. In particular the PAInteract object has to be split in two as the code involved solely with communication with the proof assistant subprocess obviously has to move to the server where the proof assistant is now located.

Other than that, all that is required is two objects to deal with the HTTP communication (illustrated on page 33), one on the client side which must encode the commands in a URL and send the request (class ServletInteract) and one on the server side - the servlet itself, class PAServlet - which handles the request, takes the necessary commands and passes them onto the server side PAInteract object (class ServletPAInt). Similarly the servlet must get the response from ServletPAInt and construct a response to return to ServletInteract.

This is an excellent example of the black box view of object oriented design, since the rest of the system is unaware of any change whatsoever - the objects on the client side continue to interact with class PAInteract in exactly the same manner.

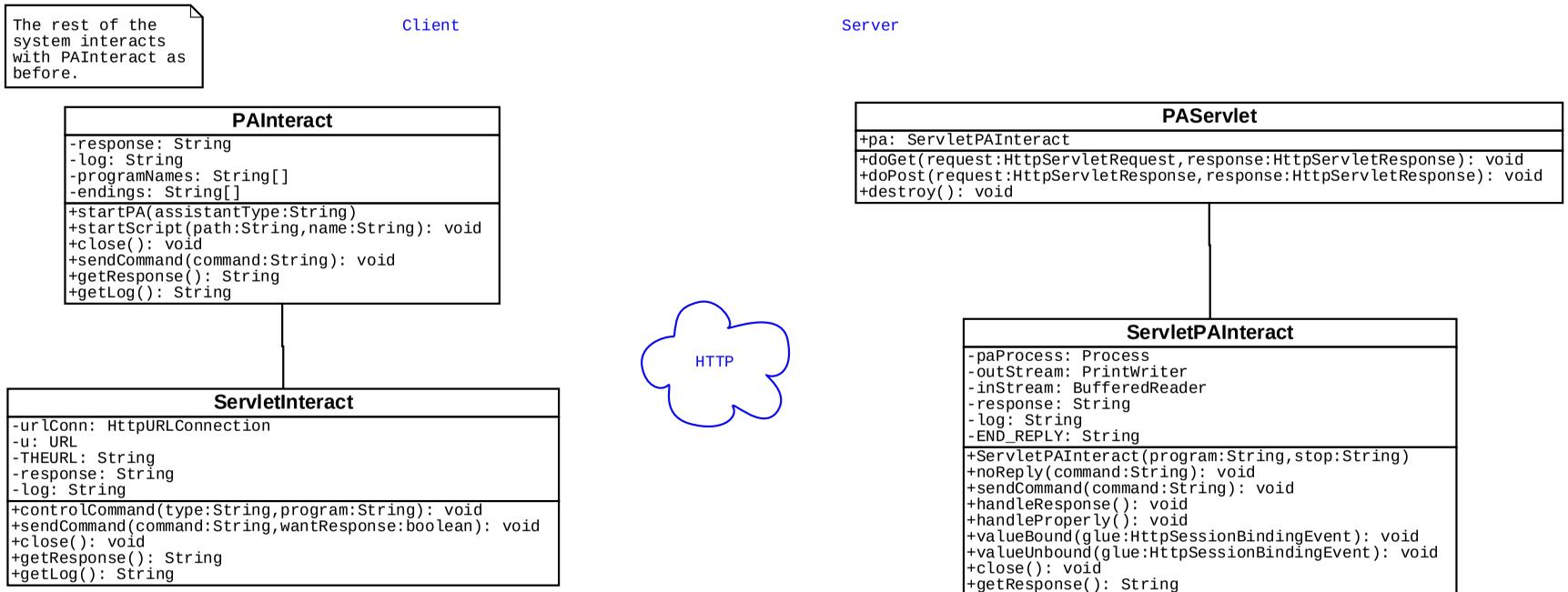
Session Tracking

Servlets also provide elegant solutions to the final two problems highlighted above - session tracking - with the use of predefined javax.servlet.http.HttpSession interface.

The servlet container uses this interface to create a session between an HTTP client and an HTTP server, a session which persists for a specified time period, across more than one connection or page request from the user. The interface itself allows servlets to view and manipulate information about a session, such as the session identifier, and importantly allows objects to be bound to sessions. Thus for each new session, representing a new user, the servlet can create a new instance of ServletPAInt and bind that object to the respective session.

When an application stores an object in or removes an object from a session,

the session checks whether the object implements HttpSessionBindingListener. If it does, the servlet notifies the object that it has been bound to or unbound from the session, allowing the ServletPAInt objects to call the necessary functions to destroy the subprocesses and tidy up the objects on the server.



A simplified class diagram representing the Client / Server architecture

Chapter 4

Implementation

The implementation process followed three distinct phases which I will document below.

Initially, I implemented the system for one proof assistant only, Isabelle. This enabled me to concentrate on the design of the script management and the GUI. The second iteration saw the system abstracted away from Isabelle to support any proof assistants implementing the necessary interface - in practice, Coq, Lego and Isabelle - various improvements were also introduced during this stage. The final, and unfinished stage, was to implement the modified version to communicate over HTTP.

This chapter should also further clarify some of the design decisions and options highlighted above.

4.1 Isabelle Replayer

Following the design discussed in 3.1 the Replayer GUI (Replayer.java), ProofScript (ProofScript.java) and Proof Assistant Interact (initially IsabelleInteract.java then PAInteract.java) objects were implemented to enable the replay of Isabelle proof scripts.

4.1.1 GUI

The GUI proved easy to implement using the predefined javax.swing components. The text display capabilities are plentiful in swing and also provide excellent scrolling support. For the file display, the output display and log display a simple JTextArea was sufficient (both from javax.swing.text.JTextComponent). Although the size of the file to be displayed was normally far greater than the typical output from the proof assistant the size of the respective displays was kept identical for esoteric reasons - the display looked unbalanced if one of the text areas was bigger than the other. All of the displays were set as uneditable as the replayer has no need for editing capabilities. It proved simple to implement the GOTO capabilities by use of the built in javax.swing.text.JTextComponent. getCaretPosition() - this returns the cursor position within the display although this obviously only works if the file representation is the same as that in the display. Unfortunately, as the display was not editable the cursor was not actually visible, so to get around this difficulty a new MouseListener was added to the file display which on a click event simply got the caret position and selected the text at that point, which served to display a pointer to the position clicked.

The menus were equally simple to implement - again swing provides excellent support for both pop-up menus (javax.swing.JComponent.JPopUpMenu) and normal menus (javax.swing.JComponent.JMenuBar) - including shortcut keys for keyboard interaction.

A button panel was constructed to contain the various navigation buttons which were kept simple but do allow for customization through adding icons.

Finally, the problem of how to view the log display was solved by adding both the log and file displays as separate panes to a JTabbedPane, making it extremely easy to switch views. Again swing allows for the addition of icons to the tab labels if so desired.

A private inner class was added to the GUI object which implemented interface ActionListener - the listener interface which reacts to action events - and was added to all the relevant buttons and menu options in order to handle all

action events upon them.

Figure 4.1 provides a screen shot of the GUI, with a pop up menu visible.

4.1.2 ProofScript

Since this deals with the Script Management - the major element of the project, what follows is a detailed investigation of how the design basics discussed in 3.3 were implemented.

To recap the ProofScript object has to handle the following: a representation of the proof file; the current location within the file; processing forwards; processing backwards; the record of commands to send; a means of sending the commands; construction of spans; and the record of spans constructed.

These problems were solved as follows:

File Representation

Since Java has extensive class libraries for String objects it sufficed to represent the file as a simple String. A String is actually held as a sequence of characters and as such the class includes methods for examining individual characters of the sequence, for comparing strings, for searching strings and for extracting substrings.

In particular it is very easy to keep track of the current position by the use of an integer as the above methods can by used with integer arguments representing a position within the string.

In this first iteration the file was selected from the local file system with the help of a javax.swing.JFileChooser which provides a simple mechanism for the user to select a file. I was however very much aware of the fact that eventually this procedure would be replaced by downloading the file from a server.

Processing Forwards

The following details the design described in 3.3.1. The ProofScript object contains a method `public void processForwards( int pos )` - a method called

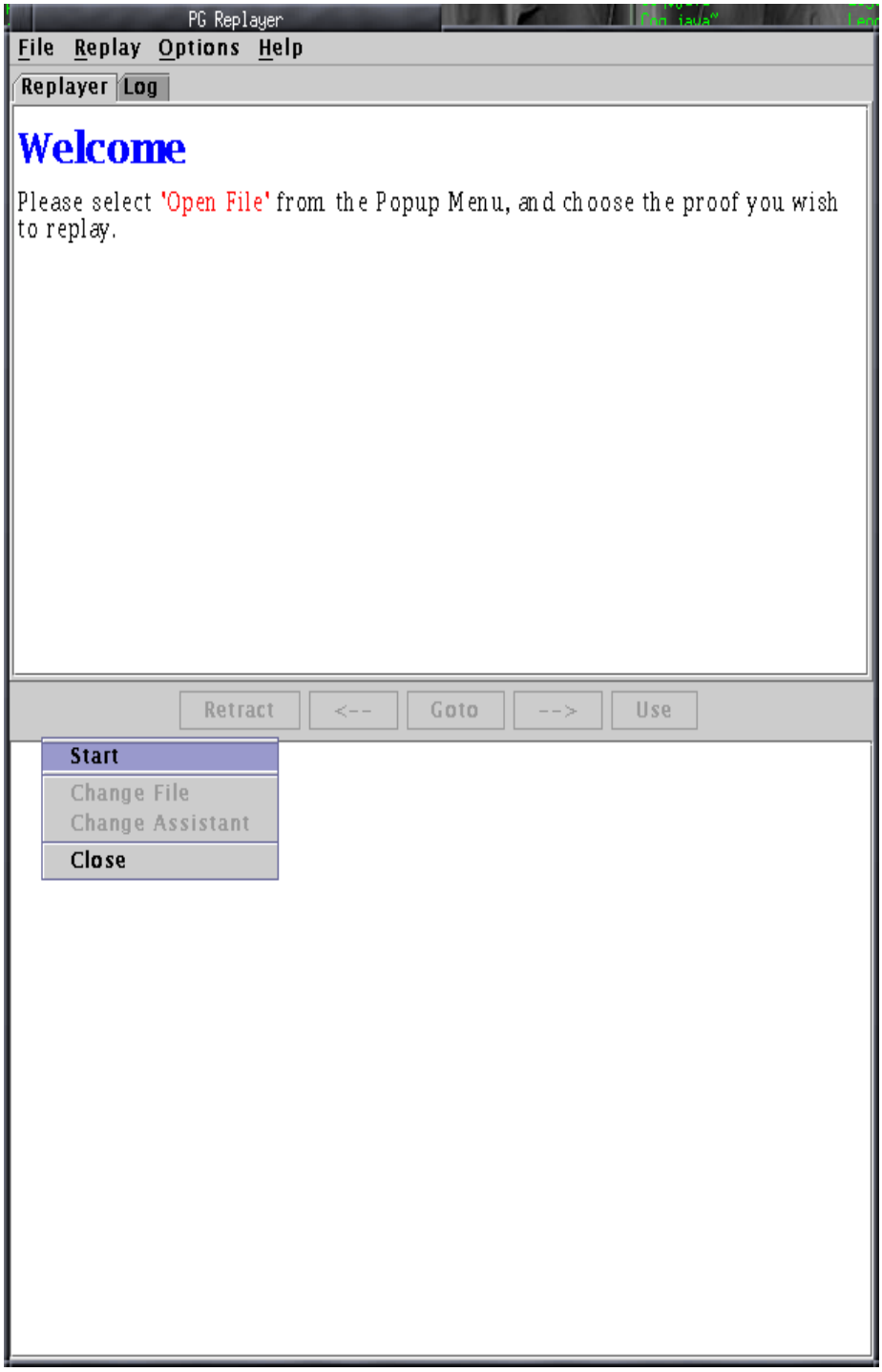


Figure 4.1: GUI Screenshot

in response to action events on the GUI requesting movement forwards in the proof to a particular position (pos); this in turn calls methods `private void buildQueue( int pos )`and `private void startQueue()`.

`buildQueue( int pos )` builds the queue of commands - held as a java.util. LinkedList. While the current position in the file is less than the position we wish to process to (ie argument pos) the next command is parsed and added to the queue, and the current position is updated.

This, obviously, requires the use of a parser which was implemented as a separate object. Again the benefits of object oriented programming are seen here as the particular implementation of the parser can be freely changed as long as the interface remains the same and it offers the same capabilities. The Parser itself holds a String representation of the file and simply searches through the string for instances of the terminal character which delimits separate commands and returns a substring of the file representation corresponding to the next command. The only difficulty here is the possibility of terminal characters appearing within comments, so this is checked for. Extra functionality was added to the Parser to simplify the construction of Spans - dealt with next.

`startQueue()` removes a command from the queue and sends them one by one (through the Proof Assistant Interact object) until the queue is empty. Having sent each command the system blocks until all of the response has been received (details in ProofAssistant Interact below), at which point another method, `private void makeSpan( String comm )`, is invoked.

The point of this function is to construct the Span object as discussed in 3.3.1, and is called every time a command is sent, once the response has been received - it is constructed here rather than when the command is parsed to ensure that the communication between the Replayer and the underlying proof assistant has been successful. In order to do this it is necessary to work out the start and end points of the new span, the type and, in the case of a Save type, the title of the proof. The type is returned from the Parser object which is handed the command for which the span is being constructed and parses

it looking for certain keywords which define its type as either Goal, Save or Command. The Span constructor is then called with this information and the new Span is pushed onto the top of a Stack (java.util) - all that is required for our last-in first-out requirements.

Process Backwards

With all of this extra information now available processing backwards is quite an easy process. ProofScript contains a `public void processBackwards( int pos )` method which, similarly, is called in response to action events on the GUI requesting movement backwards in the proof to a particular position (represented by pos); this method calls `private String calculateCommand( int pos )` which returns the required command and this is then sent to the underlying proof assistant through the Proof Assistant Interact object.

`calculateCommand( int pos )` works by popping Span objects from the Span Stack until the starting position field of the popped object is less than the position that we are processing to. The current position is reset to this starting position. Corresponding to the three different types of Undo command discussed in 3.3.1 three different actions take place when each object is popped. Firstly a counter is kept of the number of objects popped in order to issue the Undo n command (where n represents the required number) if the movement is from in-proof to in-proof; secondly, the type of each popped Span is inspected to see if it is Goal in which case we are moving from in-proof to out-proof and the KillRef command is required; finally if the type is Save then we are moving from out-proof to out-proof back through another proof which requires the Forget "id" command, where "id" is returned from the title field of the Span in question. To assist in this an inProof flag is maintained by the ProofScript object corresponding to the current proof status.

`calculateCommand( int pos )` then simply returns the correct command as a String, concatenating KillRefs and Forget "id"s if necessary, and only returning Undo n if the move backwards has remained in the same proof.

Finally, a method is included to enable the change of file during a session. This reinitializes the appropriate fields within the object and sends the necessary commands to Isabelle to alert the proof assistant to the change of file.

4.1.3 IsabelleInteract

This class, replaced in later iterations by PAInteract, represents the connection to the underlying proof assistant sub process. Java contains a class (java.lang.Process) which creates a native process and allows for connection of standard input and output through an Error, Input and OutputStream, which were connected with buffering where possible to improve efficiency.

Commands are sent to the process through method `public void sendCommand( String command )` which flushes the command down the OutputStream, and calls method `private void handleResponse()`. This method proved difficult to implement as there where considerable problems with the communication between the application and the proof assistant subprocess, and so the system is forced to block until various checks have been carried out to ensure that the end of output has been reached.

Public methods `getResponse()` and `getLog()` , which return Strings representing the last response and the whole log of interactions respectively, are also provided for the other objects to access this response.

This Object also contains methods to start Isabelle with the correct command and to initialize the proof assistant in the correct state - in particular the proof assistant needs to know the current working directory, and the file name of the proof file in order to check for theory files. Fortunately, Java's File object (java.io.File) contains the necessary methods to discover this information and the ProofScript object passes this information on.

4.2 Second Iteration: Generalization

This phase proved easy to implement following the design outlined in 3.4. The main work involved trailing through the emacs lisp files provided with the Proof General distribution to identify the correct commands and characters for the particular proof assistants.

The problems which did arise, however, continued to center on Java's interaction with the underlying proof assistants. For example, for some reason it proved impossible to force Coq to return its end of output. Unfortunately I was unable to discover the reason for this and so was forced to introduce a nasty hack in the handleResponse() method to check whether Coq was the current proof assistant; in these situations, to stop the system from blocking, the system only reads from the input stream once which occasionally means that some output is delayed until the next read (prompted by next the user interaction). A situation which very occasionally leads to a lack of synchronization between the output display and the current state of the proof assistant.

Similarly with Lego, Java seems unable to recognize some of the characters which Lego returns, resulting in the display of unsightly unknown character markers in the user displays.

The greatest benefit of this implementation is that it allows for the extension of the project with great ease. Any new proof assistants merely have to provide a *<proof assistant>* class implementing interface ProofAssistant which ensures that all the required commands and characters are returned and a *<proof assistant Filter>* class which replaces any special marker characters produced by the proof assistant by means of regular expressions with HTML equivalents to allow the display of marked up HTML in the JEditorPanes. Whereas the first of these classes is extremely rigid, the second need not actually do much at all to the output - for example, since Coq does not provide many additional characters in its output, the CoqFilter does little more than replace new lines with the HTML equivalent `<br>`.[1]

[1] The reader is directed to the Javadoc documentation in Appendix B.

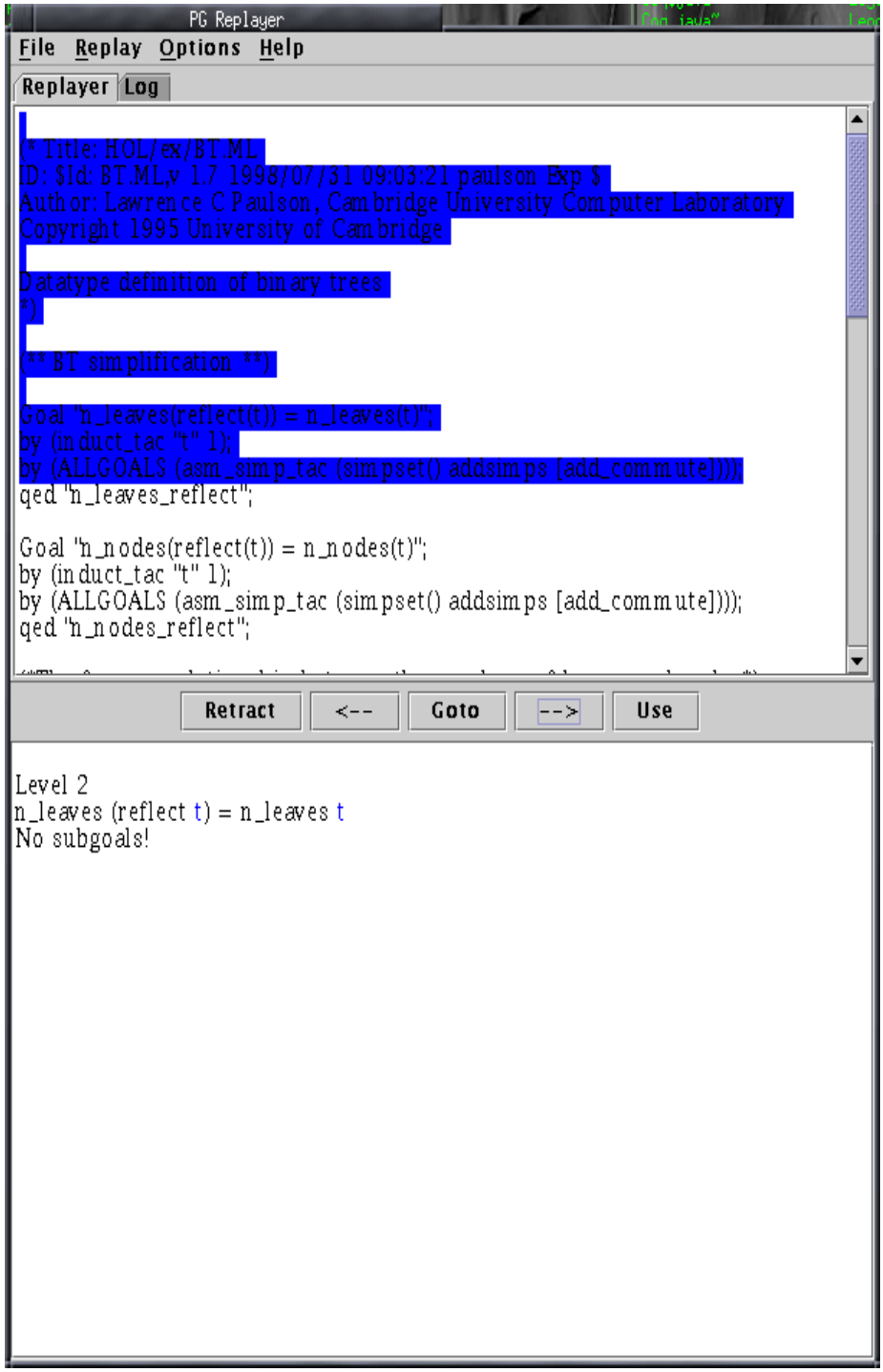


Figure 4.2: GUI Screenshot Demonstrating Script Management Features

The second iteration also introduced many practical improvements over the first. Solutions were found to the problem of focusing the display in the Scroll Pane windows upon the correct part of the overflowing proof script (by the use of javax.swing.text.JTextComponent.setCaretPosition()); a separate FileInteract class was added to the system in anticipation of the change in file access of the final version; and a FileFilter class was added to mark up the file representation of the spans and todo areas in HTML to allow for the highlighting aspect of the script management discussed in 3.3.2.

Furthermore options were added to the system to allow the user to change proof assistant as well as file, and a process was added to PAInteract (the more general class developed from IsabelleInteract) to select the correct proof assistant to start dependent on the type of file chosen by the user (for example, proof script files ending in .ML correspond to Isabelle, while those ending in .v correspond to Coq).

4.3 Servlet Implementation

Unfortunately, the third iteration, implementing the client/server architecture, remains unfinished. During this stage some severe problems were encountered, which proved extremely difficult to solve due in part to a lack of experience of server side programming and servlet technology, but mostly due simply to a lack of time.

The design outlined in 3.5 proved reasonably easy to implement, and the communication over HTTP between the class constructing the HTTP requests, ServletInteract, on the client side and the servlet dealing with the service requests and constructing the HTTP response on the server side worked perfectly.

Whereas in the second iteration PAInteract's public method sendCommand() (the method through which ProofScript sends commands to the proof assistant) simply prints the command down an output stream to the proof assistant process and then calls its own private method handleResponse() to block until all

output from the proof assistant is received, here the method sendCommand() is changed to pass the command onto ServletInteract. ServletInteract then communicates with the servlet, constructing an HTTP request and blocking the client side until the appropriate response has been received. This code from the first version of PAInteract, is then moved to the server side in class ServletPAInteract - an object constructed by the servlet for every new user session which starts and communicates with the required proof assistant process.

The session tracking mechanism also appeared to run smoothly, recognizing different user sessions and binding objects to the respective sessions with the predefined classes of the javax.servlet package. Similarly, thanks to the HttpSessionBindingListener mechanism, server resources where freed when sessions were terminated (either by time-out or browser shutdown).

Unfortunately the code involved with opening and maintaining the input and output streams with the underlying proof assistant process did not work in the server side servlet container provided by the Jakarta Tomcat framework. Strangely this was exactly the same code which worked smoothly on the client side.

A great deal of time was spent attempting to discover the cause of this problem, but unfortunately, with no servlet, or even general server expertise and with time at a premium, it proved impossible to solve. However, it should be noted that it is possible that these problems occurred because of the settings for the Jakarta Tomcat servlet container (a situation which was obviously investigated) and it is hoped that with more time available the servlet can be tested on the Proof General server.

Chapter 5

Testing & Evaluation

Due to the problems in completing the final client/server version of this project I was unable to carry out anywhere near the level of testing which I had initially intended and which the system requires before release. The issue of formal testing is quite complicated and extremely important for a GUI based application and I discuss some of these elements below, having first mentioned some of the testing I did manage to undertake.

5.1 Testing

A series of small tests were carried out after the implementation of the second stage on the stand alone application connected to proof assistants running locally on my home machine. This involved a small selection of proof scripts and four volunteers and was never intended to be anything more than a brief, initial usability test to generate feedback upon the design and functionality of the system[1]. Although the user cross section was very limited (four students - two CS, one Math and one Language - aged between 21 and 24; all same level of expertise - knowledgeable computer users but with no experience of proof

[1] In fact research by Jakob Nielson & Tom Landauer[13] suggests that five people represent the ideal number of test users and that it is far better to carry out a series of small tests than use all the resources on one large test. See *http://www.useit.com/alertbox/20000319.html*.

assistants / Proof General) this did produce some useful feedback. In particular one of the responses suggested that I had committed the cardinal sin of forgetting the user, and designing for what I know rather than what the user knows. In this case the complaint resided around the rather confusing and unnecessary distinction between changing file and changing proof assistant, where the former is selected to switch to a different file for the currently running proof assistant, while the latter is used for changing file and assistant.

This was a very fair and wholly justified criticism, although in my defense, they were in effect testing a development prototype and many of the procedures would change when the application was changed to a web-based applet (for example files would be downloaded from the server rather than the current method of file selection); the distinction between changing file and changing assistant had been made as it was my intention to present the user with a selection of files for the currently selected proof assistant and a separate selection of supported proof assistants, the selection of one of which would result in the list of files changing.

In addition to this testing, I personally tested the Replayer throughout the development of the project on a wide variety of proof scripts included with the distributions for each of the three supported proof assistants.

5.2 Validation

Validation primarily consists of checking that the correct application has been built and ensuring that it satisfies the aims of the project. Commercially this would probably involve a strict series of requirements, agreed before the start of the project, whereas here the requirements were quite vague as it was unknown at the start of the project quite how much would be achieved.

5.3 Usability Evaluation

A usability evaluation encompasses many different issues, but can be said to center upon two: whether the system has the right functionality and whether it is easy to learn. Since this project is based around a graphical user interface the single biggest factor determining successful use is the quality of the environment which the user must work in.

Topics which should be tested for include the following:

5.3.1 Learnability

How easily new users can interact meaningfully with the system. Factors here include familiarity, whether the system appears familiar to users and resembles other similar applications; predictability, whether knowledge of the system state allows the user to determine the consequences of the next action; and synthesizability, whether each action has an obvious effect or result.

5.3.2 Flexibility

The number of ways the user and system exchange information. Central to this are substitutability, offering various methods of interaction to achieve the same result, and customizability, the ability of the user to change the interface or add their own shortcuts.

5.3.3 Robustness

The ability of the system to support the users in achieving their goals. Key here are recoverability - the ability to undo errors - and responsiveness - whether the system responds in a reasonable time period.

Unfortunately, the designers familiarity with his system severely interferes with his ability to efficiently evaluate it and is one of the main reasons why outside testing is so important. Although the above factors were in the forefront

of my mind during the design and implementation phases, undoubtedly many faults would show up as a result of a further series of test.

Chapter 6

Conclusion

6.1 Achievements

This project has succeeded in the majority of its aims, although unfortunately the system is as of yet not web based. A Replayer has been produced for Proof General which allows for the replaying of proof scripts for three different proof assistants, Isabelle, Coq and Lego. All of the required script management features, including file and output highlighting, were successfully implemented.

It is indeed unfortunate that the final web-based version is not yet completed, although it is sincerely hoped that eventually the replayer will be hosted on the Proof General web site. At the end of the project there just proved to be too little time to deal with the implementation and necessary testing of a finished web-site, although a stage was reached where communication did successfully take place over HTTP.

One of the most pleasing aspects was that the design goal of producing an extendable system was successfully met. The system should prove simple to adapt to provide support for other proof assistants.

6.1.1 Improvements

Having said that most of the original objectives were met, there still remains plenty of scope for improvements to the Replayer.

There remain problems with the communication between the Java system and the underlying proof assistant process, resulting in ugly output from Lego and a lack of output termination confirmation from Coq.

More syntax highlighting can take place, which would serve to improve the clarity of the system.

A huge amount of testing remains to be done, as well as extra work on user documentation and support.

And obviously, the final web based version can be delivered.

6.2 Future Work

The greatest scope for future work lies in the integration of the Replayer with the latest stage of the Proof General project, PG Kit[1]. The Kit introduces a new architecture for Proof General, representing a collection of communicating components centered upon a new protocol for interactive electronic proof, PGIP, consisting of small XML documents. PGIP comes with an associated markup language PGML, used for markup of output from the proof assistant to be displayed to the user.

Fortunately, because of the extendibility of the project design it should be possible to support this by the implementation of another proof assistant instance.

6.3 Summary

While it is disappointing not to have completed a web based version of the Replayer, on the whole I have been pleased with this project and the hope

[1] [1] and [2] provide a discussion of the work in progress on this latest development.

remains that I will have the satisfaction of seeing the Replayer running on the Proof General website.

The project as a whole suffered somewhat from being rather open ended; although this may not have represented a problem for a more experienced programmer, it caused me great difficulties early on as I was unaware of precisely which direction I should be heading in and as a result a great deal of time was lost exploring details which in the end were unnecessary - in particular the XML technologies surrounding the PGIP discussed above.

Personally however, I have taken a great deal from what has been a very enjoyable experience and have learnt a great deal concerning the problems of project management and system implementation.

Appendix A

Code

The following pages list the code for the classes making up the main application.

Appendix B

Javadoc

This is the documentation created by the javadoc tool for the classes representing the Proof Assistant Objects and their dependents.

Bibliography

[1] Aspinall. Proof General Kit White Paper, (LFCS, Feb 2000)

[2] Aspinall. Protocols for Interactive e-Proof, (LFCS, May 2000)

[3] Proof General Manual (version 3.2) (LFCS, May 2000)

[4] Bertot & Thery. A Generic Approach to Building User Interfaces for Theorem Provers (Journal of Symbolic Computation, 1998, 25(7) 161-194)

[5] java.sun.com - The Source for Java Technology, *http://www.java.sun.com/*

[6] Java 2 SDK Documentation, *http://java.sun.com/j2se/1.2/docs/index.html*

[7] The Java Tutorial, *http://java.sun.com/docs/books/tutorial/*

[8] Flanagan. Java Examples In a Nutshell, (O'Reilly, 1997)

[9] Flanagan. Java In a Nutshell, (O'Reilly, 1997)

[10] Deitel & Deitel. Java - How to Program (Third Edition), (Prentice Hall ISBN 0-13-012507-5)

[11] Deitel, Deitel & Nieto. Internet & World Wide Web - How to Program, (Prentice Hall ISBN 0-13-016143-8)

[12] Principles of Good GUI Design, *http://axp16.iie.org.mx/Monitor/v01n03/ar_ihc2.htm*

[13] Jakob Nielson's Column on Web Usability, *http://www.useit.com/alertbox/*

[14] Oberlander, CS4 Human Computer Interaction Course Notes, (The University of Edinburgh, 1999)

[15] The Jakarta Project: Tomcat, *http://jakarta.apache.org/tomcat/*

[16] Servlets.com, *http://www.servlets.com/*

[17] Hypertext Transfer Protocol, *http://www.w3.org/Protocols/HTTP/*

[18] Regular Expressions for Java, *http://www.cacas.org/java/gnu/regexp/*

[19] XML.org The XML Industry Portal, *http://www.xml.org/*

[20] Argo UML, *http://argouml.tigris.org/*

[21] Gnome Office - Dia, *http://www.gnome.org/gnome-office/dia.shtml*

[22] LyX - The Document Processor, *http://www.lyx.org/*